White Paper

James Coleman
Performance Engineer
Perry Taylor
Performance Engineer
Intel Corporation

Hardware Level IO Benchmarking of PCI Express*

December, 2008

321071

Executive Summary

Understanding the PCI Express* performance capabilities of embedded design is critical in selecting a solution that supports the IO requirements for the intended usage model. Theoretical bandwidth data is not sufficient since it is simply a calculation of frequency and bus width, not taking into account protocol overhead, bus/link efficiency, platform latency or bandwidth scaling across multiple IO devices.

This paper addresses how to set PCI Express* performance targets and how to collect hardware-level measurements. It begins with an overview of Intel® architecture and PCI Express architecture. Following these overviews, we focus on setting PCI Express performance targets, PCI Express measurement methodology, tuning, and interpreting results. This paper will help the reader understand what is required to collect meaningful PCI Express performance data and what the results mean.

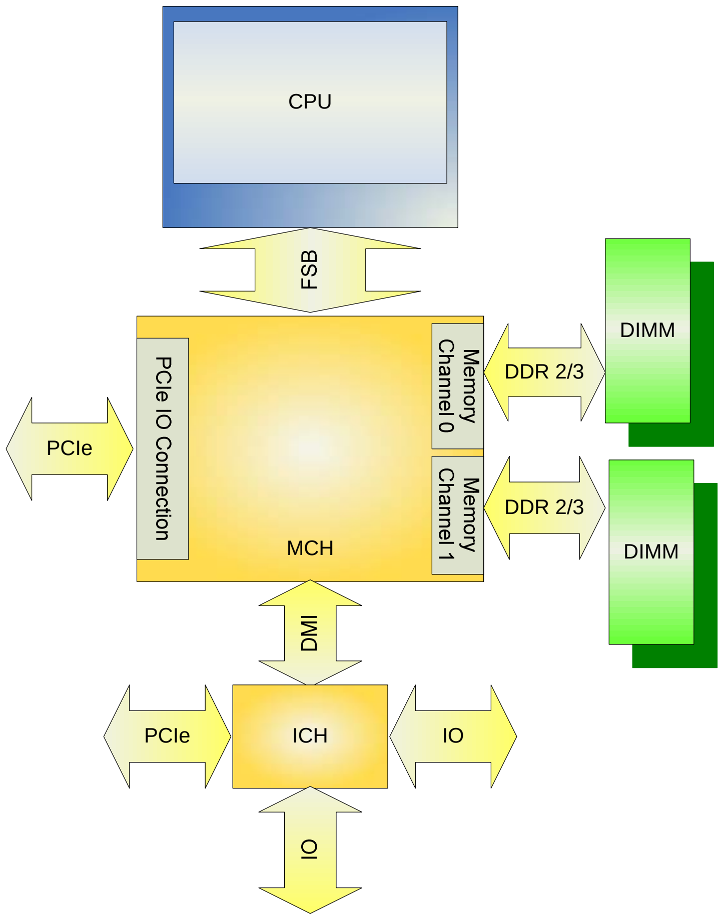
This paper addresses how to set PCI Express* performance targets and collect hardware-level measurements.

Contents

Intel® Architecture Overview

This section will cover a brief overview of the Intel® architecture. For more information on this topic, refer to "Introduction to Intel Architecture – The Basics" (http://download.intel.com/design/intarch/papers/321087.pdf). Intel® architecture can be broken down into two main categories, components (silicon) and interfaces connecting the silicon. The components of interest are the Central Processor Unit (CPU), Memory Controller Hub (MCH) and I/O Controller Hub (ICH). The interfaces of interest are the Front Side Bus (FSB) or Intel® QuickPath Interconnect (QPI), memory interface, PCI Express* (PCIe*) and Direct Media Interface (DMI). Figure 1 shows a generic Intel® architecture diagram.

Figure 1. Intel® Architecture Block Diagram



CPU

At a high level, the CPU is the brain of the system where all execution occurs and all other components of IA support the CPU. The CPU consists of the execution units and pipelines, caches and FSB unit. Entire books have been written detailing CPU architecture, but for this paper there is an important concept to understand relating to cacheable memory. CPU caches are filled one full cache line at a time. This means that if one byte is required by an execution unit, the CPU will fetch an entire cache line (64 bytes) into the CPU cache. All accesses to cacheable memory are done so in cache line size quantities. This behavior with cacheable memory is consistent across all interfaces and components.

MCH

If the CPU is the brain, then the MCH is the heart of Intel® architecture. The MCH routes all requests and data to the appropriate interface. It is the connecting piece between the CPU, memory and IO. It contains the memory controller, FSB unit, PCI Express* ports, DMI port, coherency engine and arbitration.

ICH

The ICH provides extensive IO support for peripherals: USB, audio, SATA, SuperIO, LAN as well as the logic for ACPI power management, fan speed control and reset timing. The ICH connects to the MCH via the DMI interface, a proprietary "PCI Express*-like" interface.

FSB

The FSB is a parallel bus that enables communication between the CPU and MCH. The FSB consists of 32 to 40 address bits and strobes, 64 data bits and strobes, 4 request bits and side band signals. Data is quad-pumped, allowing data transfer of 32 bytes per bus clock. Transactions are broken into phases, such as request, response, snoop and data. For this paper it is important to know that the FSB is utilized for all coherent IO traffic. An IO transaction to system memory will produce a read transaction on the FSB to snoop (lookup) CPU cache for conflicting addresses.

Intel® QuickPath Interconnect (QPI)

QPI is a serial bus enabling communication between components. Like PCI Express*, it is packet-based and has a proprietary protocol.

PCI Express Overview*

PCI Express* (PCIe*) represents the third generation, high-performance IO bus, with the first generation being ISA and the second PCI. The software architecture of PCI Express is fully backwards compatible with PCI, allowing a legacy OS (Operating System) with no knowledge of PCIe to work seamlessly with PCIe hardware. This software compatibility is immediately evident by the bus, device, and function addressing scheme of PCIe devices, which is the same as PCI. This paper assumes the reader has a reasonable knowledge of PCI system architecture. This paper is not intended to provide a comprehensive overview and/or introduction into PCIe but rather focuses on the benchmarking methodology used by Intel. For more information regarding PCIe and its differences with respect to PCI, the reader is encouraged to refer to the PCIe specification or one of the many technical books written on this subject.

Despite its backwards compatibility, PCIe has several advanced features which, to be fully realized, require software, namely OS support. However, the greatest advantage of PCIe over PCI is drastically improved bandwidth. This benefit is realized even with a legacy OS. The following table shows the bandwidth of different versions of PCI compared to PCIe.

Table 1. IO Bus Peak Bandwidths

Bus Type	Peak Bandwidth
PCI (66 MHz)	266 MB/s
PCI-X (133 MHz)	533 MB/s
PCIe x16 (Gen 1)	8 GB/s
PCIe x16 (Gen 2)	16 GB/s

These substantial improvements in IO bandwidth are realized by moving from a shared parallel bus (PCI), which limits the speed of the clock to a serial, point-to-point, bi-directional interconnect that allows simultaneous traffic in both directions and operates at much higher frequencies (2.5 GHz for Gen 1 and 5 GHz for Gen 2). This move not only allows the clock frequency to increase (and with it the bandwidth) but also significantly reduces the pin count (thus reducing the packaging cost of components and the cost of routing traces on a motherboard).

Moving to a serial bus necessitated a packet-based protocol rather than a bus cycle-based protocol. Despite the additional overhead of a packet-based protocol, the real world efficiency of a PCIe link is well over 90% whereas the efficiency of a PCI bus is on the order of 50% – 60%. The effects of this overhead on actual bandwidth versus the peak theoretical bandwidth shown in the above table are discussed in more detail in the Setting Expectations section.

PCIe* represents a layered protocol consisting of the Transaction Layer, the Data Link Layer, and the Physical Layer. Only the Transaction Layer and Transaction Layer Packets (TLPs) will be discussed in this paper.

PCIe uses a buffer tracking scheme to ensure the receiver always has sufficient buffers for what the transmitter is sending. This scheme is referred to as flow control. Flow control has the transmitter keep track of the number of free buffers the receiver has. At the time the link is trained, each end of the point-to-point connection tells the other how big its receive buffer is. This information is represented as flow control credits.

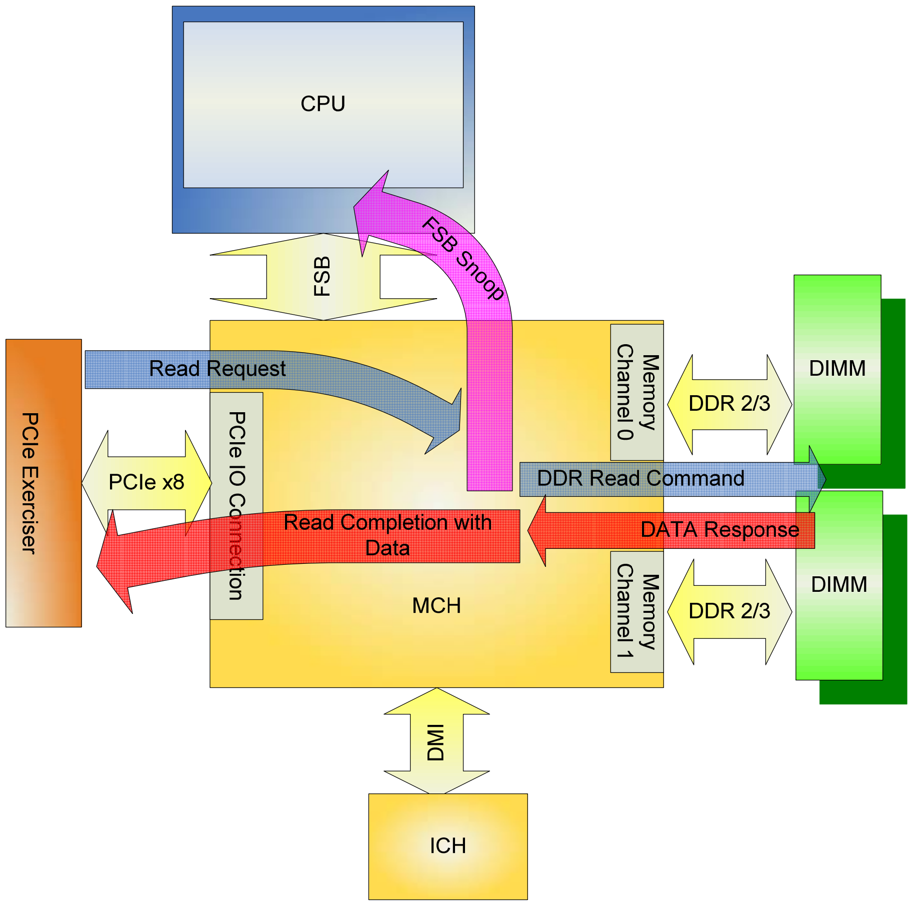
Each time a packet is transmitted, the transmitter decrements its flow control credits based on the amount of data sent. Whenever a receiver processes a received packet and frees the buffers associated with it, the receiver then notifies the transmitter of the freed buffers via a special flow control update packet. The transmitter then increments its flow control credits.

The majority of PCIe transactions are either reads from system memory or writes to system memory. The following two sections give an overview of each of these transactions.

PCI Express* Read Transaction Overview

When a PCIe* attached endpoint device needs to access data stored in system memory, it generates a read transaction on the PCIe link that will be serviced by the root complex. The endpoint issuing the read from memory determines from which address the data will be read as well as the amount of data to read. In addition to the address and the size of data to be read, the endpoint is also responsible for determining if the memory being accessed needs to be coherent with the CPU. For example, a buffer set up by the driver which is only accessed by the PCIe device would not need to be kept coherent with the CPU since the CPU would never access the data. However, a buffer used for passing data and commands between the CPU and the PCIe device would need to be kept coherent. The device driver writer is responsible for determining what regions of system memory accessed by the PCIe device need to be kept coherent with the CPU. Figure 2 shows the data paths traversed by a read request and its accompanying completion.

Figure 2. Read Request Flow



PCIe* Read Request Packet Assembly

Once the endpoint has determined the address from which to read, the amount of data to be read, and if the read is to be coherent with the CPU, then the PCIe read request packet can be assembled. The PCIe core of the device constructs a Transaction Layer Packet (TLP) based on the details of the read transaction needed by the endpoint.

PCIe* Read Request Packet Transmission

Once the packet has been fully assembled and is ready to be transmitted, the PCIe* core logic must ensure that the root complex has sufficient buffers to receive the packet. The PCIe core logic uses the number of flow control credits currently available to determine if the root complex has sufficient buffers. If not enough flow control credits are currently available, then the packet is held until the PCIe device receives a flow control update from the root complex, processes this update, and increments the flow control credits, allowing the waiting packet to be transmitted to the root complex. Once it receives the packet, the root complex checks it for errors using the TLP Digest, if present, as well as the CRC (Cyclical Redundancy Check). Once the packet's integrity has been validated, the root complex forwards the packet on to the MCH (Memory Controller Hub).

Read Request Routing

If the PCIe device issuing the read request was an ICH (I/O Controller Hub) attached device, then the read request would first traverse the point to point interconnect between the north bridge and the south bridge to the MCH (see Figure 1).

Snooping of the Data for the Read Request

Once in the MCH, the read request will generate a snoop on the FSB (Front Side Bus) if the packet is coherent, (i.e., the NS bit is clear).

Note: Even if a request specifies No Snoop, it may still be snooped on the FSB. However, a packet without the NS bit set always will generate a snoop transaction on the FSB.

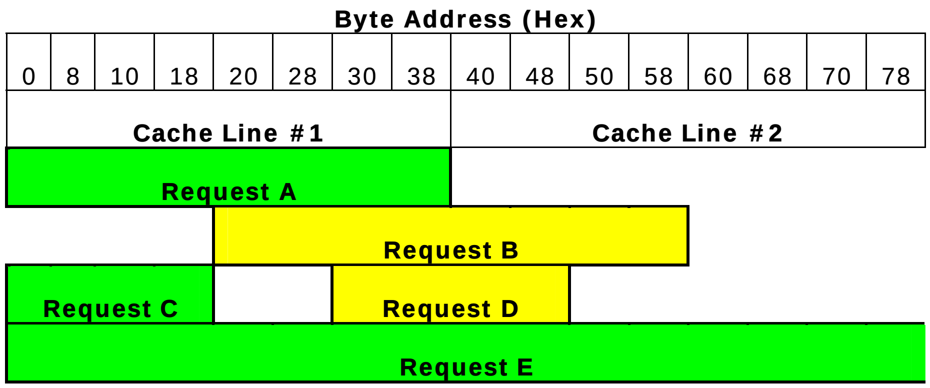
During a snoop, the MCH asks all attached CPUs to check their caches for the presence of a particular cache line. The CPUs respond, indicating if the line is present in their cache and if it has been modified. If the snooped cache line has been modified by a CPU, then the CPU will send the contents of the cache line to the MCH so that system memory can be updated. The MCH will also use this data to fulfill the read request from the PCIe device.

If the cache line has not been modified by a CPU, then the MCH will get the data from system memory to fulfill the request. All operations performed by the MCH, both snooping and actual data retrieval from system memory, are performed on whole cache lines. Therefore, a read of less than 64 bytes will result in a full cache line's being read and possibly snooped. Additionally, a request that is not 64-byte aligned (i.e., with a hex address not ending in 0x00, 0x40, 0x80, 0xC0) will incur an additional snoop and/or system memory access.

Table 2 shows five requests of various sizes and with varying alignments.

- Request A is an aligned 64-byte request, which will generate one snoop and one read from system memory.
- Request B is also a 64-byte request but is unaligned and will result in two cache lines' being snooped and two cache lines' being read from system memory.
- Request C is a 32-byte aligned request and will generate a snoop for a full cache line as well as a full cache line read from system memory.
- Request D is a 32-byte unaligned access and will result in two full cache lines' being snooped and in two full cache lines' being read from system memory.
- Request E is a 128-byte aligned access, resulting in two cache lines' being snooped and two cache lines' being read from system memory.

Table 2. Request Cache Alignment



PCIe* Read Completion Packet Assembly

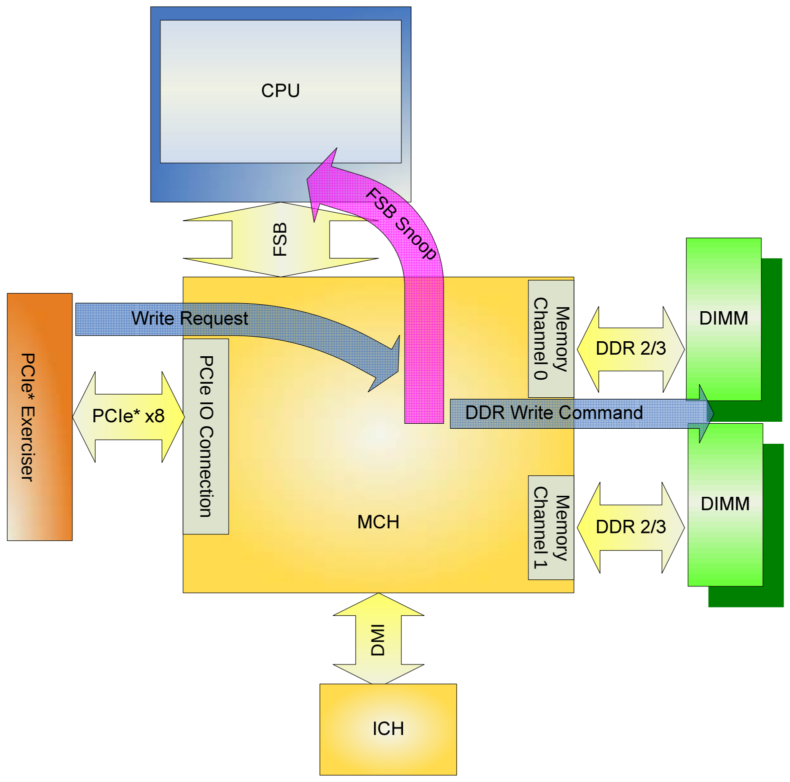
Once the root complex has the data for the read request, it can begin assembling the read completion packet. In the case of either an unaligned request or an aligned request larger than 64 bytes, the data will arrive for each cache line, causing the root complex to generate a read completion packet per cache line received. However, if the platform has read completion combining enabled, then the root complex can opportunistically hold cache lines received and coalesce them into a read completion packet with a payload of the size specified by the max payload size. In the case of small (64 byte or less), unaligned requests, the read completion coalescing would result in a single read completion packet per request rather than two read completion packets, greatly improving the efficiency of the PCIe* link.

Once the root complex has assembled the read completion packet, having coalesced any data, it checks the flow control credits currently available and transmits the packet to the PCIe device only when sufficient flow control credits exist. In the case of a PCIe device with small buffers (a few flow control credits), using a large payload size and enabling read completion combining would result in lower performance as the root complex will wait for all the buffers to be cleared before it sends a single, large completion packet. In this case, the root complex would achieve better performance by sending several smaller read completion packets, thereby avoiding the latency involved in draining the buffers and updating the flow control credits.

PCI Express* Write Transaction Overview

When a PCIe* attached endpoint device needs to write data to system memory, it generates a write transaction on the PCIe link that will be serviced by the root complex. The endpoint issuing the write to memory determines in which address the data should be stored as well as the contents to be stored. In addition to the address and the data to be written, the endpoint is also responsible for determining if the memory being written needs to be coherent with the CPU. For example, a buffer set up by the driver which is only accessed by the PCIe device would not need to be kept coherent with the CPU since the CPU would never access the data. However, a buffer used for passing data and commands between the CPU and the PCIe device would need to be kept coherent. The device driver writer is responsible for determining what regions of system memory accessed by the PCIe device need to be kept coherent with the CPU. Figure 3 shows the data paths traversed by a write transaction and its accompanying cache coherency snoop.

Figure 3. Write Transaction Flow



PCIe* Write Packet Assembly

Once the endpoint has determined to which address to write, the data to be written, and if the write is to be coherent with the CPU, then the PCIe write packet can be assembled. The PCIe* core of the device constructs a TLP based on the details of the write transaction needed by the endpoint. Once the TLP header has been assembled in hardware, it is passed down to the data link layer, where it is encapsulated inside of a Data Link Layer Packet (DLLP). Then the DLLP is passed down to the physical layer for further encapsulation and transmission.

PCIe* Write Packet Transmission

Once the packet has been fully assembled and is ready to be transmitted, the PCIe* core logic must ensure that root complex has sufficient buffers to receive the packet along with its accompanying data payload. The PCIe core logic uses the number of flow control credits currently available to determine if the root complex has sufficient buffers. If not enough flow control credits are currently available, then the packet is held until the PCIe device receives a flow control update from the root complex, processes this update, and increments the flow control credits, allowing the waiting packet to be transmitted to the root complex. Once it receives the packet, the root complex checks for errors using the TLP Digest, if present, as well as the CRC. Once the packet's integrity has been validated, the root complex forwards the write on to the MCH.

Write Routing

If the PCIe* device issuing the write was an ICH attached device, then the write would first traverse the point to point interconnect between the north bridge and the south bridge to the MCH.

Snooping of the Data for the Write

Once in the MCH, the write request will generate a snoop on the FSB if the packet is coherent, i.e., the NS bit is clear (Note: Even if a write specifies No Snoop, it may still be snooped on the FSB. However, a packet without the NS bit set always will generate a snoop transaction on the FSB).

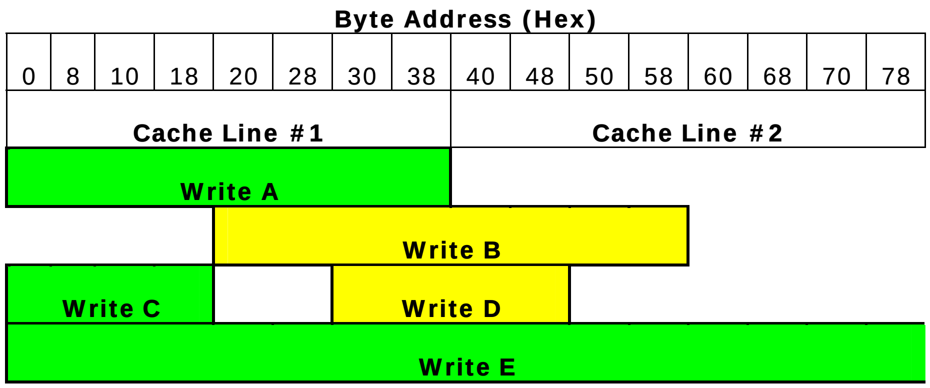
During a snoop, the MCH tells all attached CPUs to check their caches for the presence of a particular cache line, and, if found, to invalidate that line. A CPU that has modified the cache line will send the contents of the cache line to the MCH so that system memory can be updated. The MCH will also use the data in the write packet to update system memory.

All operations performed by the MCH, both snooping and writes to system memory, are performed on whole cache lines. Therefore, a write of less than 64 bytes (partial write) will result in a full cache line read. The data read will be merged with the data to be written. Then the cache line will be written back to system memory. Additionally, a request that is not 64-byte aligned (i.e., with a hex address not ending in 0x00, 0x40, 0x80, 0xC0) will incur these read-merge writes on both ends of the system memory access.

Table 3 shows five requests of various sizes and with varying alignments.

- Request A is an aligned 64-byte write, which will generate one snoop and one write to system memory.
- Request B is also a 64-byte request but it is unaligned and will result in two cache lines' being snooped and in two cache lines' being read from system memory. The data from the write will be merged with each cache line, and then they will each be written back, resulting in two reads and two writes even though only one cache line's worth of data needs to be written.
- Request C is a 32-byte aligned write and will generate a snoop for a full cache line as well as a full cache line read from system memory. The data to be written will be merged and then the full cache line will be written back to system memory.
- Request D is a 32-byte unaligned access and will result in two full cache lines' being snooped and two full cache lines' being read from system memory. Then the data to be written will be merged with both cache lines, and both cache lines will be written back to system memory.
- Request E is a 128-byte aligned write, resulting in two cache line snoops and two cache line writes to system memory.

Table 3. Request Cache Alignment



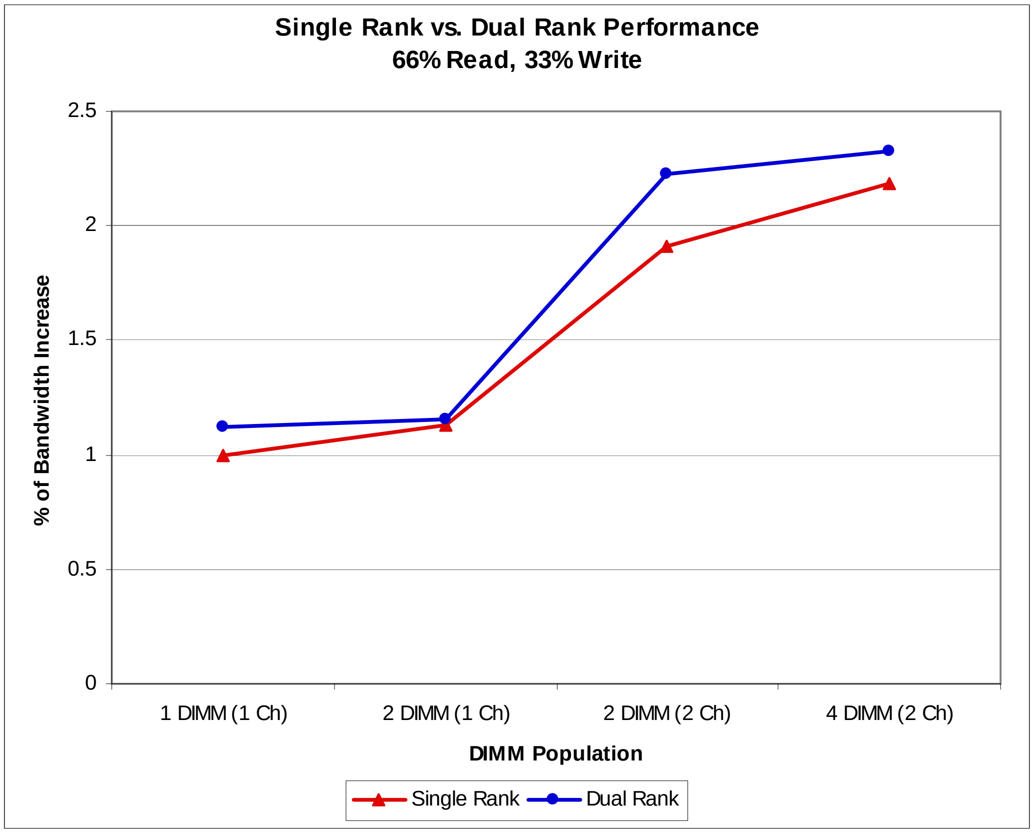
Setting PCI Express Performance Expectations*

A question that is often asked is "Why am I not getting theoretical bandwidth?" Before any actual performance testing is undertaken, one should take some thought on setting realistic performance targets. Unfortunately, there is no one formula to set expectations since all factors of interest change per design. However, this section will help the reader in understanding the factors that affect bandwidth and the setting realistic performance expectations.

CPU to Memory Bandwidth

Before proceeding to IO performance measurements, we recommend that users first verify memory performance via CPU to memory tests. This is a quick way to validate for the optimal price/performance configuration of the platform and to check system performance against available targets. Note that Intel® Memory Controller Hubs support multiple memory frequencies, DRAM arrangements, CPUs and CPU interface frequencies. All of these variables have an impact on the memory performance. Figure 4 illustrates this impact on a sample platform including dual rank vs. single rank DIMMs in arrangements of: 1 DIMM – 1 channel, 2 DIMMs – 1 channel, 2 DIMMs – 2 channel, 4 DIMMs 2 channel.

Figure 4. Example Memory Performance with Varied Configurations



PCI Express* Efficiency & Protocol Overhead

For the following example, we will address one x8 PCI Express* link running at generation 1 speed.

Theoretical Bandwidth

Before proceeding to protocol overhead, it is worth mentioning how theoretical bandwidth is calculated and why it can never be reached.

A x8 can place eight symbols on a lane per symbol time. These eight symbols represent 64 bits of data, or eight bytes. Since a symbol time is 4ns we can calculate the bandwidth as:

1,000,000,000(ns)/4(ns) * 8(B) = 2,000,000,000 bytes/second (or 2GB/s)

Since PCI Express* can send and receive concurrently, the theoretical bandwidth is said to be 4GB/s for a x8 lane. This assumes that every symbol time contains 8 full bytes of "useful data" in both transmit and receive lanes. Obviously, it is not possible to transmit 100% useful data since PCI Express is packet-based and data must be wrapped within a packet.

Table 4. Theoretical Bandwidth Table

Link Width	Theoretical Bandwidth (Bi-directional)
x1	512 MB/s
x2	1 GB/s
x4	2 GB/s
x8	4 GB/s
x16	8 GB/s

Protocol Overhead

A packet containing data consists of framing (start and end), sequence number, header, data, digest, and link cyclical redundancy check (LCRC). Below we show the byte size breakdown of a packet containing 64 bytes of data:

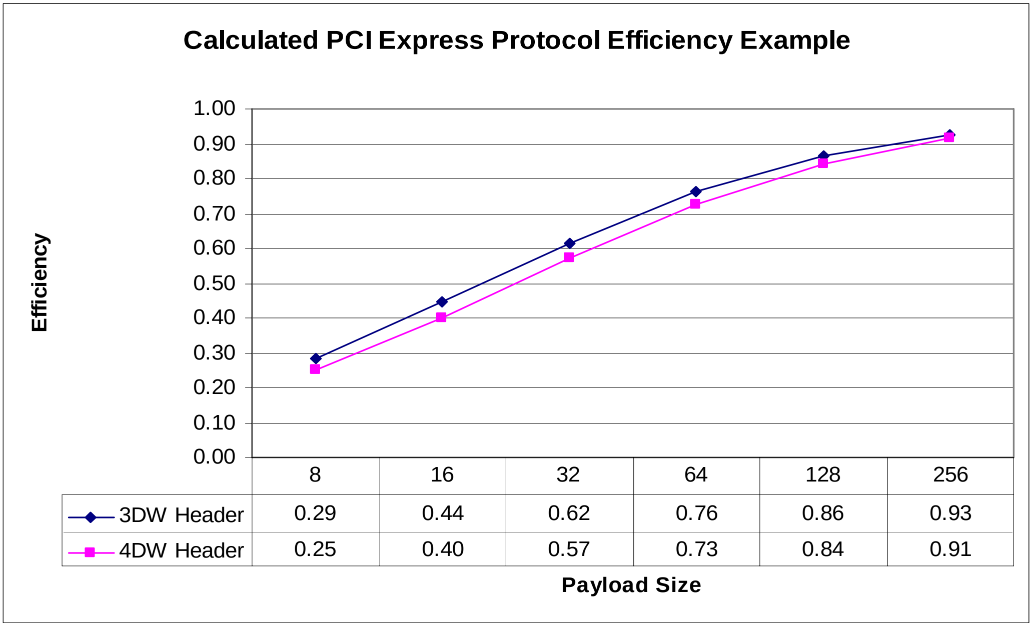
- Start of Packet (STP) – 1 Byte
- Sequence Number – 2 Bytes
- Header – 12 Bytes (16 Bytes if >4GB address)
- Data – 64 Bytes
- LCRC – 4 Bytes
- END – 1 Byte

To transmit these 64 bytes of useful data, we must pack it in 20 bytes of wrapping, yielding a maximum efficiency of 76%, or 3.04GB/second, for our x8 link.

Efficiency

We can now calculate efficiency based on the protocol overhead and the payload size of the traffic. Keep in mind that this does not represent achievable bandwidth since it assumes that all lanes are transmitting TLPs 100% of the time with no bubbles or DLLPs. Efficiency gives us the upper bound of achievable bandwidth per link and is required for setting realistic performance targets.

Figure 5. Calculated PCI Express* Efficiency Example



	8	16	32	64	128	256
◆ 3DW Header	0.29	0.44	0.62	0.76	0.86	0.93
■ 4DW Header	0.25	0.40	0.57	0.73	0.84	0.91

Latency

Latency plays an important role in PCI Express* performance and should be considered in relationship to bandwidth. For example, if the round trip latency for a read of 128 bytes is 400ns, then the read bandwidth would be 2.5MB/second (assuming one outstanding transaction). If the latency is increased to 800ns, then the read bandwidth decreases to 1.25MB/second, a one to one relationship.

For this reason, a PCI Express device placed directly to the MCH will have higher bandwidth potential than if connected to the ICH (or any other bridging device). The latency delta between the MCH connection and ICH connection will vary based on the chipset.

Outstanding Transaction Count

As seen above, latency has a one to one relationship to bandwidth when there is only one outstanding transaction at a time. When setting performance expectations, the workload under test must be examined for the number of outstanding transactions. If only one transaction is in flight at a time, setting performance expectations is simple math. However, as the work load increases the number of outstanding transactions, the impact of latency is reduced or hidden. Intel recommends that end usage modules utilize this transaction buffering because as the outstanding transaction count reaches the maximum, the lane efficiency increases and bandwidth can come closer to the maximum efficiency calculated above. Again, note that the maximum efficiency bandwidth will not be achievable, but getting close to 10% is reasonable.

Aggregate Bandwidth Scaling

Once expectations are set for a single link, then we can consider concurrent links, or aggregate bandwidth scaling. The first step is to simply sum the bandwidth expected/measured of one link. This gives the ideal or perfect scaling target (which may or may not be achievable). The ability of the platform to scale PCI Express bandwidth across multiple links depends greatly on the architecture and configuration. In general, the scaling will begin to level at some point as some platform resource limit is reach. This limiting factor may be memory bandwidth, FSB address bandwidth, QPI trackers or flow control (as a result of latency).

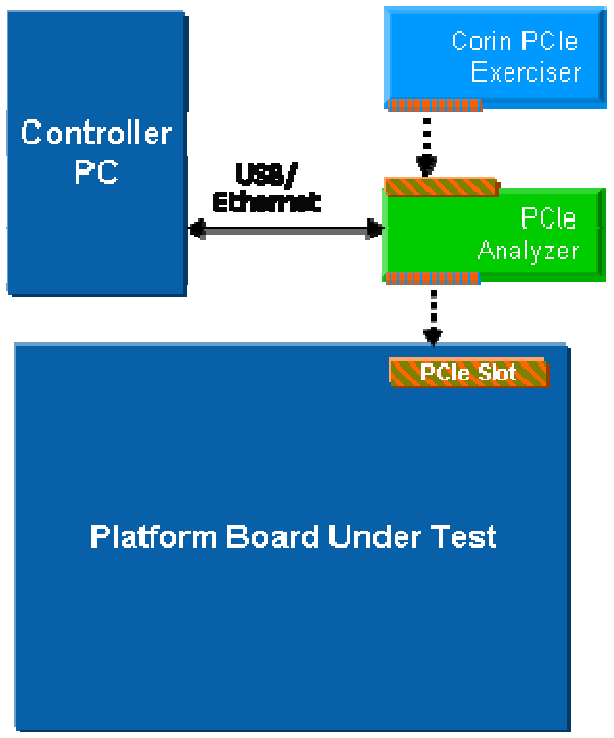
Synthetic PCI Express* Benchmarking Methodology

Synthetic PCI Express benchmarking can be broken down into two parts:

1. Generate Traffic.
2. Observe Traffic.

In this section, we first discuss synthetic PCI Express* traffic generators (exercisers) and cover traffic options and recommendations. After this, PCI Express analyzers will be discussed. Note that Intel does not endorse any specific PCI Express exerciser or analyzer and that this is not intended as a user manual for an analyzer/exerciser, but rather a focus on what type of traffic can be of interest.

Figure 6. Exerciser / Analyzer Configuration Example



Generate / Exercise

Exerciser Overview

A PCI Express* exerciser is a necessary piece of equipment for PCI Express performance testing. An exerciser is a tool that can be programmed with a specific traffic mix and run in a loop, generating a continuous and known traffic pattern. This is done by placing the exerciser's card into the slot under test and configuring the exerciser through a controller system running the exerciser's interface program.

The exerciser removes typical bottlenecks of end devices such as network and disk throughput/latency, allowing testers to better analyze the performance capabilities of a platform.

Some examples of exercisers are the Agilent N5309A* and the LeCroy PETrainer SummitZ2. More information on these exercisers can be found at www.agilent.com and www.lecroy.com.

Request Type Variations

PCI Express* exercisers offer the ability to drive any type of transaction to any destination address. For performance testing, the most interesting transactions are reads and writes to memory. Interrupts may also be of interest for measuring interrupt latency as well as read/write to another PCI Express endpoint for testing peer-to-peer performance.

End users should create performance tests with 100% read traffic, 100% write traffic, and 50% read 50% write with max outstanding transactions. These three data points will provide best case read, write, and bi-directional throughput. Any read/write ratio traffic patterns can be created and should be tested based on end user usage models.

Request Size Variations

PCI Express protocol allows transaction request sizes from one byte to 4096 bytes. Typical request sizes for performance testing are 64, 128, 256, 512, 1024, 2048 and 4096 bytes.

Snoop and No Snoop

The recommendation is to test both snoop and no snoop PCI Express throughput. Under heavy system loads, the no snoop throughput may be higher than the snooped throughput as FSB/QPI command bandwidth is consumed for each IO request to snoop the processor cache. On the opposite side, the throughput may be higher for no snoop on an idle system where the processors are entering sleep states. If the processor enters a sleep state, it must wake up before responding to snoop requests, adding latency to the transaction.

Note that most real-world IO traffic is coherent, or snooped. It should also be noted that the chipset reserves the right to ignore the no snoop bit and treat it as a snooped transaction.

Alignment

For best performance, all transaction requests should be aligned on cache line boundaries (64 bytes).

Test unaligned requests if a specific usage model requires unaligned transactions. Unaligned requests are not optimized for performance and consume additional memory bandwidth. For example, if IO performs an eight byte memory read at address 2000003Ch, 128 bytes will be read from system memory (cache line 20000000h and 20000040h).

Observation/Analyzer

Protocol Analyzer Overview

A PCI Express* analyzer is the second half of the exerciser and is required to measure PCI Express throughput/latency. An analyzer functions via a PCI Express interposer card, capturing all transactions in flight on the link. The analyzer, like the exerciser, is programmed through a controller system using the analyzer interface software. The analyzer can be programmed to trigger on specific events such as TLP reads, writes, DLLP flow control, acknowledgements (ACK) and/or on specific addresses.

Once triggered, the analyzer captures all packets in flight for a duration of time and then assembles them in a user-friendly format.

Analyzers are available from companies such as Agilent and LeCroy.

Measuring Bandwidth

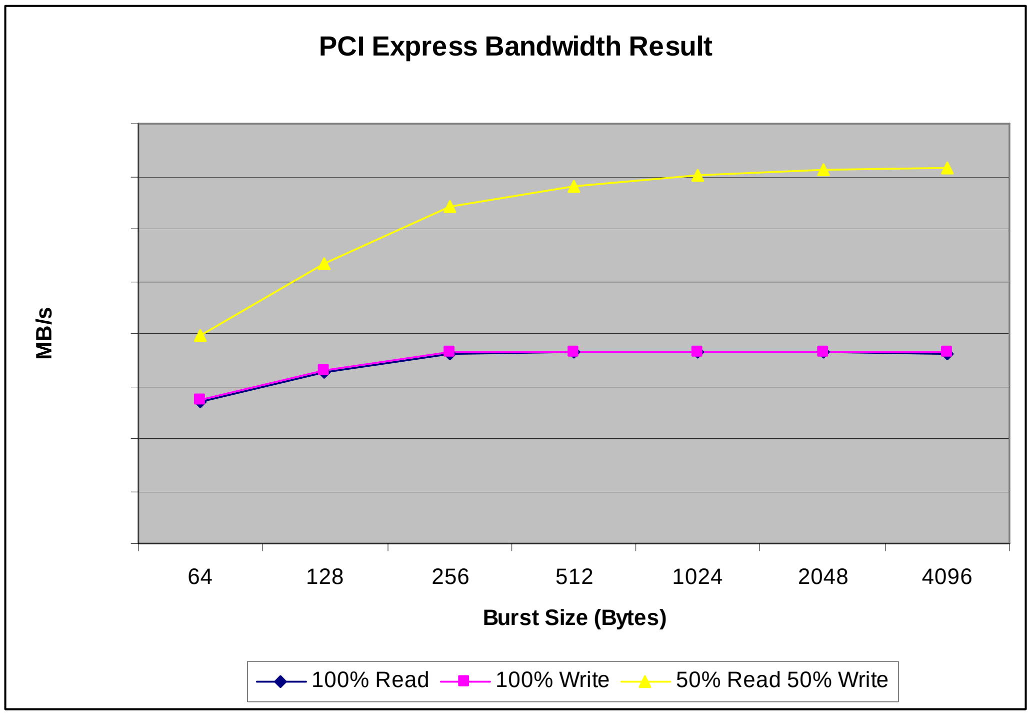
Some analyzers have a built in function that will analyze the trace captured and report the average MB/second. If this function is not available, export the trace to a text file and create a script or program to calculate the bandwidth.

Keep in mind that bandwidth can be reported as 10^6 or 2^{10}. In general, Intel reports bandwidth as 1KB=1000B.

Record the bandwidth for each transaction type of interest, such as:

- Reads – 64, 128, 256, 512, 1024, 2056, 4096 bytes
- Writes – 64, 128, 256, 512, 1024, 2056, 4096 bytes
- Read/Write – 64, 128, 256, 512, 1024, 2056, 4096 bytes
- Figure 7 is an example test result.

Figure 7. Sample Bandwidth Result



Measuring Latency

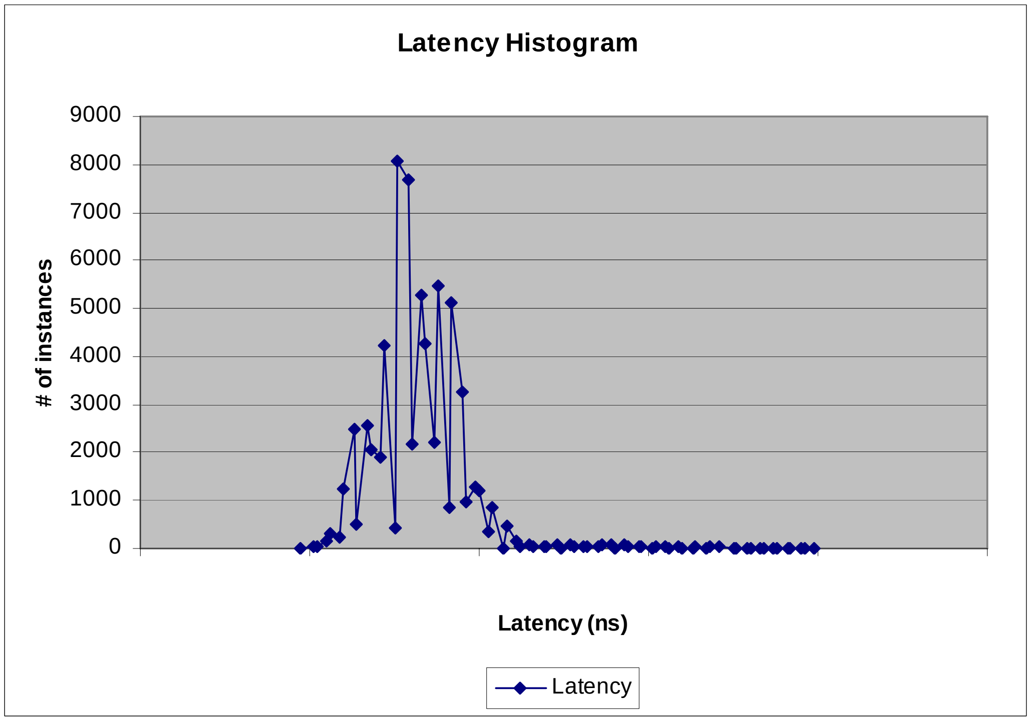
It is not possible to measure write latency since it is a posted transaction. One can measure the time from the TLP write to the ACK, but this does not represent the entire flight time of the transaction to the end destination.

To measure idle read latency, the exerciser must issue only one read request outstanding at a time. Set the analyzer to trigger on the specific address of the upstream read request. The analyzer will then display the upstream read as well as the downstream completion and the time difference is calculated by the analyzer GUI.

Measuring loaded latency is done by running the exerciser with multiple outstanding transactions and triggering on any one of these read requests. Use the Tag ID field to match the proper read and completion packets. Again, the analyzer can be set to show the time delta between the two packets. Keep in mind that latency results will vary depending on the state of the IO/memory queues, memory page, and even the processor state. For this reason, multiple samples should be observed.

A more complete way to measure the latency is to export the capture to text and create a parse routine to create a histogram of latencies. The result will produce something of a bell curve as shown below. The X-axis is the latency value, and the Y-axis represents how many transactions in the trace had that latency.

Figure 8. Sample Latency Result



Measuring Aggregate Bandwidth

The above method addresses performance testing of a single PCI Express* link, but what if we are driving multiple links? Using multiple analyzers is possible, but in most cases the tester will need to collect multiple samples, moving the analyzer to each slot under test, and then summing the results measured at each slot. This is time consuming and tedious, but one cannot assume that measuring one interface, then multiplying that result by the number of slots under test is accurate.

Finding Bottlenecks

Why isn't the PCI Express* throughput higher?

Analyzers are critical to finding PCI Express bottlenecks when higher throughput is expected. When debugging performance, take note of the number of outstanding transactions, latency, flow control, and rhythms in the traffic flow timing (rhythmic gaps). Gaps are often produced by end device latency, not turning around flow control credits quickly enough, by platform latency, inadequate outstanding requests, or by poor tuning (see next section). Many of these behaviors can be corrected with software and hardware tuning.

It is useful to use the analyzer to find the beginning point of performance failure. Often, when traffic initially begins to flow, there are no problems. At some point, however, gaps begin to appear. For this reason, it is important to trigger on the first transaction of the workload under analysis. If the trace is collected after the traffic has reached a steady state, then it may not be possible to root-cause the performance issue.

Tuning Parameters

There are a few, readily available tuning knobs related to PCI Express performance. This section details them with recommended settings. Unfortunately, the registers are not in any set location and will be at different addresses on different IO devices and chipsets. To locate the registers, locate the External Design Specification (EDS) for the architecture and find the bit fields mentioned.

Max Read Request Size

Max read request size determines the largest read request that a device can issue. It should be set to 4096 bytes for best performance. Allowing larger read requests upstream will reduce protocol overhead on the upstream link and improve performance downstream by increasing the chances for opportunistic coalescing by the chipset.

Max Payload Size

Max payload size determines the maximum number of bytes allowed in any TLP, and, in general, should be set to the max value supported. Max payload size has a direct impact on PCI Express efficiency as seen by Figure 5.

However, some ultra-low power chipsets can have a reduced buffer set, resulting in less flow control. Under this scenario, it is more efficient to reduce the max payload size to the lowest possible value.

Coalesce Settings

Coalesce features and technology vary per chipset architecture and are defined in the external design specification. In general, coalescing should be enabled for best downstream efficiency. Coalescing allows the chipset to opportunistically combine cache lines to issue completions larger than 64 bytes (i.e. 128 bytes, 256 bytes). Without coalescing, all completions will be limited to a maximum payload of 64 bytes.

Interpreting and Using the Results

Latency

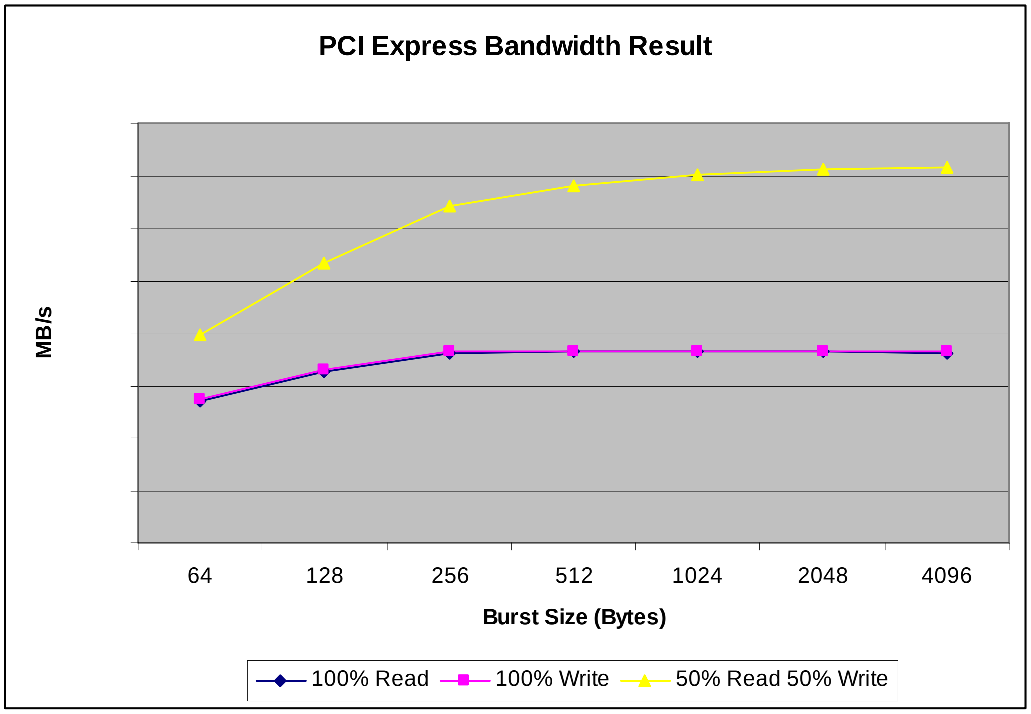
Interpreting latency results is straightforward because any slot produces a single latency number that can be compared directly with any other slot from any system under test. The lower the latency is, the better the performance potential of that PCI Express* link.

This data can be used to help determine where PCI Express devices are to be placed on a board design. For best performance, place demanding IO devices in slots that have the lowest latency (slots directly connected to MCH or CPU) and only place low demand IO devices in slots with higher latency (slots connected through a bridging device such as ICH).

Bandwidth

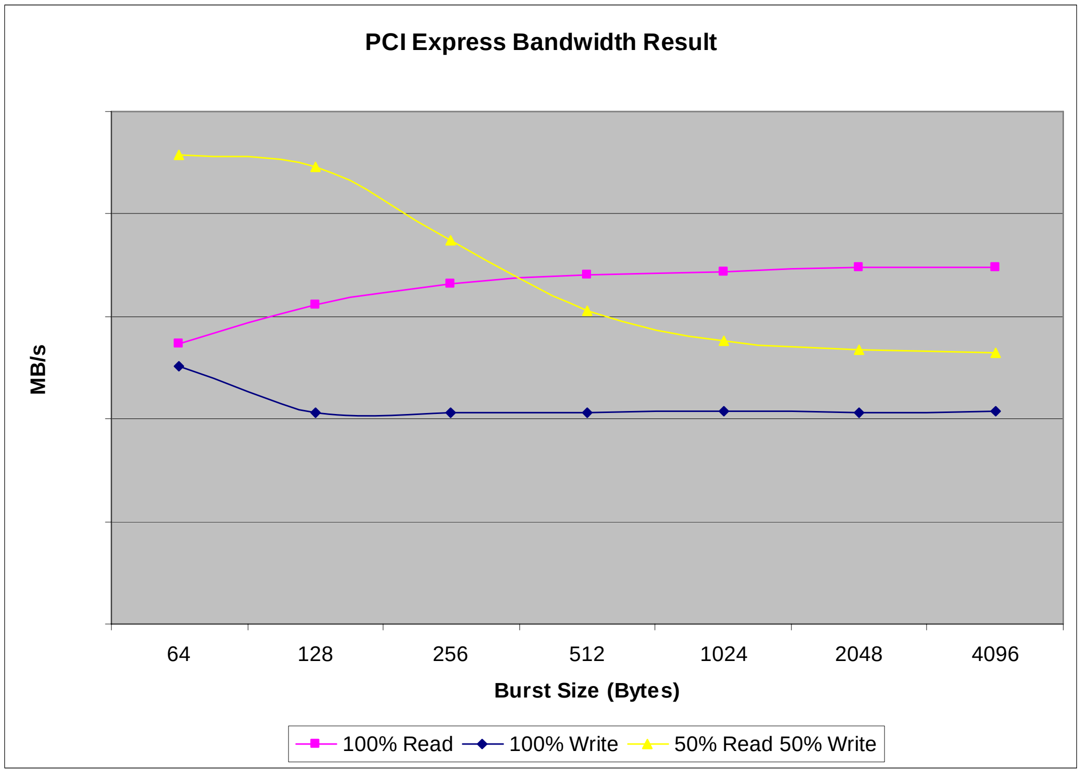
Bandwidth measurements of a single link can be used to help fine tune hardware settings and driver behavior. For example, we will consider two different architectures, A and B.

Figure 9. Architecture A Bandwidth Result



Architecture A bandwidth results show that hardware tuning should be configured to allow the largest transactions possible, setting the highest value possible to max payload size, max read request size, and coalescing. Likewise, software drivers should also take advantage of the higher efficiency of large transactions and attempt to drive large transaction requests.

Figure 10. Architecture B Bandwidth Result



Architecture B bandwidth results show that hardware tuning should not be configured the same as architecture A, which favors large transaction sizes (unless our device traffic is a vast majority of inbound reads). For mixed traffic, the best bandwidth is at 64 bytes, and hardware/software tuning should reflect this.

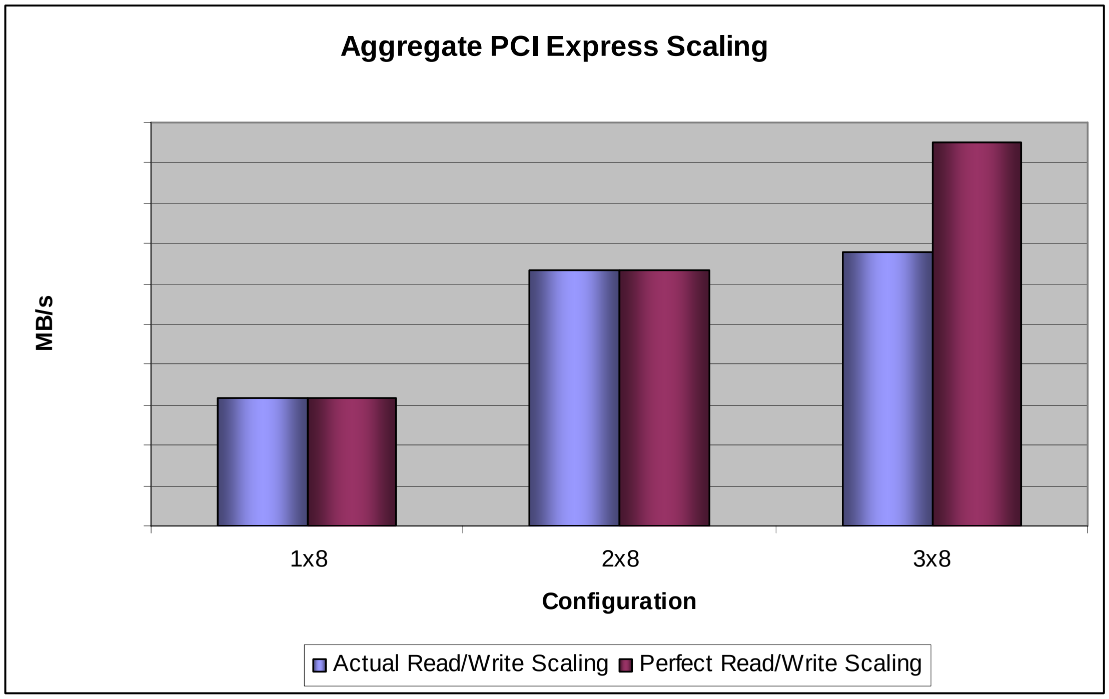
Aggregate Bandwidth

Aggregate bandwidth results tell us how much total IO throughput can be expected when multiple slots are populated. It is important to have this data when configuring the platform with IO devices or when selecting a platform for IO usage models. For example, if a usage model requires two or three high demand x8 PCI Express devices with perfect bandwidth scaling, one should verify that the architecture can support this bandwidth.

This data is also useful when an IO workload is not performing as expected. Aggregate bandwidth testing proves that the PCI Express and memory system can sustain the desired bandwidth, or, it can also prove that the workload has reached the bandwidth limitations of the platform.

Figure 11 shows that the platform under test scales perfectly from one to two x8 links, but scaling from two to three slots is 71%.

Figure 11. Aggregate PCI Express* Scaling Example



Conclusion

Hardware-level performance testing of the PCI Express* interfaces with proper targets is critical in understanding the fundamental IO performance capability of a platform. Collecting PCI Express performance data following the explained methodology enables design architects to make IO placement decisions for best performance. It also ensures that a product can deliver the performance required for the intended usage model. Hardware-level PCIe performance data can also aid debug efforts in root-causing IO solutions with lower than expected throughput.

Authors

James Coleman is a performance engineer with Intel Embedded Communications Group.
Perry Taylor is a performance engineer with Intel Embedded Communications Group.

Acronyms

PCI	Peripheral Connect Interface
PCIe*	PCI Express*
FSB	Front Side Bus
CPU	Central Processing Unit
DMI	Direct Media Interface
MCH	Memory Controller Hub
ICH	I/O Controller Hub
DIB	Dual Independent Bus
MB/s	Megabyte per Second
GB/s	Gigabyte per Second
MT/s	Megatransfers per Second
DIMM	Dual In-line Memory Module

321071